Birla Institute of Technology and Science, Pilani
CS F212 Database Systems
Lab No # 7

****

1 Procedures

The following SELECT statement returns all rows in the table customer from the Sakila database.

```
select customer.customer_id, customer.first_name,
customer.last_name
from sakila.customer;
```

If you want to save this query on the database server for execution later, one way to do it is to use a stored procedure.

The following *CREATE PROCEDURE* statement creates a new stored procedure that wraps the query above:

```
DELIMITER $$
CREATE PROCEDURE GetCustomers()
BEGIN
select customer.customer_id, customer.first_name,
customer.last_name
from sakila.customer;
END$$
DELIMITER ;
```

By definition, a stored procedure is a segment of declarative SQL statements stored inside the MySQL Server. In this example, we have just created a stored procedure with the name GetCustomers().

Once you save the stored procedure, you can invoke it by using the CALL statement:

```
CALL GetCustomers();
```

And the statement returns the same result as the query.

- ☞ The first time you invoke a stored procedure, MySQL looks up for the name in the database catalog, compiles the stored procedure's code, place it in a memory area known as a cache, and execute the stored procedure.

- ☞ If you invoke the same stored procedure in the same session again, MySQL just executes the stored procedure from the cache without having to recompile it.
- ☞ A stored procedure can have parameters so you can pass values to it and get the result back. For example, you can have a stored procedure that returns customers by country and city. In this case, the country and city are parameters of the stored procedure.
- ☞ A stored procedure may contain control flow statements such as IF, CASE, and LOOP that allow you to implement the code in the procedural way.
- ☞ A stored procedure can call other stored procedures or stored functions, which allows you to modularize your code.

MySQL stored procedures advantages

- ☞ You can avoid having to store all your SQL code in files. Stored procedures are stored in the database itself, so you never have to search through files to find the code you want to use.
- ☞ You can execute a stored procedure as often as you like from any machine that can connect to the database server.
- ☞ If you have a report that needs to be run frequently, you can create a stored procedure that produces the report. Anyone who has access to the database and permission to execute the stored procedure will be able to produce the report at will. They don't have to understand the SQL statements in the stored procedure. All they have to know is how to execute the stored procedure.
- ☞ Reduce network traffic: Stored procedures help reduce the network traffic between applications and MySQL Server. Because instead of sending multiple lengthy SQL statements, applications have to send only the name and parameters of stored procedures.
- ☞ Centralize business logic in the database: You can use the stored procedures to implement business logic that is reusable by multiple applications. The stored procedures help reduce the efforts of duplicating the same logic in many applications and make your database more consistent.
- ☞ Make database more secure: The database administrator can grant appropriate privileges to applications that only access specific stored procedures without giving any privileges on the underlying tables.

MySQL stored procedures disadvantages

- ☞ Resource usages: If you use many stored procedures, the memory usage of every connection will increase substantially. Besides, overusing a large number of logical operations in the stored procedures will increase the CPU usage because the MySQL is not well-designed for logical operations.
- ☞ Troubleshooting: It’s difficult to debug stored procedures. Unfortunately, MySQL does not provide any facilities to debug stored procedures like other enterprise database products such as Oracle and SQL Server.

- ☞ Maintenances: Developing and maintaining stored procedures often requires a specialized skill set that not all application developers possess. This may lead to problems in both application development and maintenance.

Create procedure syntax:

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

proc_parameter:
    [ IN | OUT | INOUT ] param_name type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:

    Valid SQL routine statement
```

- ☞ *CREATE* statements are used to create a stored procedure. That is, the specified routine becomes known to the server. By default, a stored routine is associated with the default database. To associate the routine explicitly with a given database, specify the name as *db_name.sp_name* when you create it.
- ☞ Each parameter is an *IN* parameter by default. To specify otherwise for a parameter, use the keyword *OUT* or *INOUT* before the parameter name.
- ☞ The *routine_body* consists of a valid SQL routine statement. This can be a simple statement such as *SELECT* or *INSERT*, or a compound statement written using *BEGIN* and *END*.
- ☞ The *COMMENT* characteristic is a MySQL extension, and may be used to describe the stored routine.
- ☞ The *LANGUAGE* characteristic indicates the language in which the routine is written. The server ignores this characteristic; only SQL routines are supported.
- ☞ A routine is considered "deterministic" if it always produces the same result for the same input parameters, and "not deterministic" otherwise. If neither *DETERMINISTIC* nor *NOT DETERMINISTIC* is given in the routine definition, the default is *NOT DETERMINISTIC*.
- ☞ *CONTAINS SQL* indicates that the routine does not contain statements that read or write data. This is the default if none of these characteristics is given explicitly.
- ☞ *NO SQL* indicates that the routine contains no SQL statements.
- ☞ *READS SQL DATA* indicates that the routine contains statements that read data (for example, SELECT), but not statements that write data.

- ☞ *MODIFIES SQL DATA* indicates that the routine contains statements that may write data (for example, INSERT or DELETE).
- ☞ *The SQL SECURITY* characteristic can be DEFINER or INVOKER to specify the security context; that is, whether the routine executes using the privileges of the account named in the routine DEFINER clause or the user who invokes it.
- ☞ The *DEFINER* clause specifies the MySQL account to be used when checking access privileges at routine execution time for routines that have the *SQL SECURITY DEFINER* characteristic.

Example: - Now redefine the GetCustomer() procedure.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `GetCustomers`()
    READS SQL DATA
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'Stored procedure example'
BEGIN
     select customer.customer_id, customer.first_name,
     customer.last_name
     from sakila.customer;
END$$
DELIMITER ;
```

Example 1: - Write a MySQL procedure to show student list for semester. Name of semester is given as input.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE
`list_semester`(IN sem_name VARCHAR(10))
    READS SQL DATA
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'Smester wise list of student, Input -
Semester name and Output – Student count'
BEGIN
select student.name from university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = sem_name);
END$$
```

```
DELIMITER ;
```

Now you can call the above procedure as follows.

```
call list_semester('Spring');
```

Example 2: - Write a MySQL procedure to show student list for semester. Name of semester is given as input and count the number of students in that semester.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE
`list_semester_and_count`(IN sem_name VARCHAR(10), out
student_count int)
    READS SQL DATA
    DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT 'Smester wise list of student, Input -
Semester name.'
BEGIN
select student.name from university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = sem_name);
    select count(student.name) into student_count from
university2.student
where student.ID IN
(select takes.id from university2.takes
where takes.semester = sem_name);
END$$
DELIMITER ;
```

Now you can call the above procedure as follows.

```
call list_semester_and_count('Fall', @st_count);
select @st_count;
```

The following example demonstrates how to use an INOUT parameter in the stored procedure.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `SetCounter`(
INOUT counter INT,     IN inc INT )
    DETERMINISTIC
    SQL SECURITY INVOKER
```

```
    COMMENT 'INOUT example.'
BEGIN
SET counter = counter + inc;
END$$
DELIMITER ;
```

In this example, the stored procedure SetCounter() accepts one INOUT parameter (counter) and one IN parameter (inc). It increases the counter (counter) by the value of specified by the inc parameter.

These statements illustrate how to call the SetSounter stored procedure:

```
SET @counter = 1;
CALL SetCounter(@counter,1); -- 2
CALL SetCounter(@counter,1); -- 3
CALL SetCounter(@counter,5); -- 8
SELECT @counter; -- 8
```

- ☞ An IN parameter passes a value into a procedure. The procedure might modify the value, but the modification is not visible to the caller when the procedure returns.
- ☞ An OUT parameter passes a value from the procedure back to the caller. Its initial value is NULL within the procedure, and its value is visible to the caller when the procedure returns.
- ☞ An INOUT parameter is initialized by the caller, can be modified by the procedure, and any change made by the procedure is visible to the caller when the procedure returns.
- ☞ For each OUT or INOUT parameter, pass a user-defined variable in the CALL statement that invokes the procedure so that you can obtain its value when the procedure returns.

Note: - MySQL permits routines to contain DDL statements, such as CREATE and DROP. MySQL also permits stored procedures (but not stored functions) to contain SQL transaction statements such as COMMIT. Stored functions may not contain statements that perform explicit or implicit commit or rollback.

2 Functions

A stored function in MySQL is a set of SQL statements that perform some task/operation and return a single value. It is one of the types of stored programs in MySQL. When you will create a stored function, make sure that you have a CREATE ROUTINE database privilege. Generally, we used this function to encapsulate the common business rules or formulas reusable in stored programs or SQL statements.

The stored function is almost similar to the procedure in MySQL, but it has some differences that are as follows:

- ☞ The function parameter may contain only the IN parameter but can't allow specifying this parameter, while the procedure can allow IN, OUT, INOUT parameters.
- ☞ The stored function can return only a single value defined in the function header.
- ☞ The stored function may also be called within SQL statements.
- ☞ It may not produce a result set.

Thus, we will consider the stored function when our program's purpose is to compute and return a single value only or create a user-defined function.

The syntax of creating a stored function in MySQL is as follows:

```
CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:

    Valid SQL routine statement
```

Example: - Consider Sakila database, find out that whether a DVD rental is overdue? If yes, then count overdue days.

We create to stored function, 1) to check overdue, 2) to count days.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`overdue`(return_date DATE) RETURNS varchar(3) CHARSET
utf8mb4
    DETERMINISTIC
BEGIN
     DECLARE sf_value VARCHAR(3);
```

```
        IF curdate() > return_date
            THEN SET sf_value = 'Yes';
        ELSEIF  curdate() <= return_date
            THEN SET sf_value = 'No';
        END IF;
     RETURN sf_value;
    END$$
DELIMITER ;

DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`count_days`(return_date DATE) RETURNS int
    DETERMINISTIC
begin
     declare days INT;
        IF curdate() > return_date
            THEN SET days = DATEDIFF(curdate(),
return_date);
        ELSEIF  curdate() <= return_date
            THEN SET days = 0;
        END IF;
        return days;
end$$
DELIMITER ;
```

To call these functions as follows.

```
select rental.customer_id, rental.rental_date,
rental.return_date, curdate(),
overdue (rental.return_date) as 'Is Overdue?',
count_days(rental.return_date) as 'No. Days'
from sakila.rental;
```

customer_id	rental_date	return_date	curdate()	Is Overdue?	No. Days
130	2005-05-24 22:53:30	2020-10-24 22:04:30	2021-03-12	Yes	139
459	2005-05-24 22:54:33	2020-10-26 19:40:33	2021-03-12	Yes	137
408	2005-05-24 23:03:39	2020-10-30 22:12:39	2021-03-12	Yes	133
333	2005-05-24 23:04:41	2020-11-01 01:43:41	2021-03-12	Yes	131
222	2005-05-24 23:05:21	2020-10-31 04:33:21	2021-03-12	Yes	132
549	2005-05-24 23:08:07	2020-10-25 01:32:07	2021-03-12	Yes	138

3 Cursors

MySQL supports cursors inside stored programs. The syntax is as in embedded SQL. Cursors have these properties:

1. Asensitive: The server may or may not make a copy of its result table
2. Read only: Not updatable
3. Nonscrollable: Can be traversed only in one direction and cannot skip rows

In MySQL, Cursor can also be created. Following are the steps for creating a cursor.

1. Declare Cursor

```
DECLARE cursor_name CURSOR FOR
Select statement;
```

2. Open Cursor

```
Open cursor_name;
```

3. Fetch Cursor

```
FETCH cursor_name INTO variable_list;
```

4. Close Cursor

```
Close cursor_name;
```

Note: - Cursor declarations must appear before handler declarations and after variable and condition declarations.

Example: - Suppose there is two tables (t1, and t2) containing two fields item and price from two different vendors. We wish to find out the lowest price for each item and store it in another table. We can do this by using stored procedure and cursor.

```
CREATE PROCEDURE low_price()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT item, price FROM t1;
  DECLARE cur2 CURSOR FOR SELECT price FROM t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
  OPEN cur1;
  OPEN cur2;
```

```
  read_loop: LOOP
    FETCH cur1 INTO a, b;
    FETCH cur2 INTO c;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF b < c THEN
      INSERT INTO t3 VALUES (a,b);
    ELSE
      INSERT INTO t3 VALUES (a,c);
    END IF;
  END LOOP;
  CLOSE cur1;
  CLOSE cur2;
END;
```

4 Nested call of stored routines

Function *inventory_in_stock* return that whether a film is in stock or not. And procedure *film_in_stock* return the count of stock.

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` FUNCTION
`inventory_in_stock`(p_inventory_id INT) RETURNS
tinyint(1)
    READS SQL DATA
BEGIN
    DECLARE v_rentals INT;
    DECLARE v_out     INT;

    #AN ITEM IS IN-STOCK IF THERE ARE EITHER NO ROWS IN
THE rental TABLE
    #FOR THE ITEM OR ALL ROWS HAVE return_date POPULATED

    SELECT COUNT(*) INTO v_rentals
    FROM rental
    WHERE inventory_id = p_inventory_id;

    IF v_rentals = 0 THEN
      RETURN TRUE;
    END IF;
```

```
    SELECT COUNT(rental_id) INTO v_out
    FROM inventory LEFT JOIN rental USING(inventory_id)
    WHERE inventory.inventory_id = p_inventory_id
    AND rental.return_date IS NULL;

    IF v_out > 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
END$$
DELIMITER ;

DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE
`film_in_stock`(IN p_film_id INT, IN p_store_id INT, OUT
p_film_count INT)
    READS SQL DATA
BEGIN
     SELECT inventory_id
     FROM inventory
     WHERE film_id = p_film_id
     AND store_id = p_store_id
     AND inventory_in_stock(inventory_id);

     SELECT FOUND_ROWS() INTO p_film_count;
END$$
DELIMITER ;
```

To know more on –

- Restrictions on Stored Programs
  https://dev.mysql.com/doc/refman/8.0/en/stored-program-restrictions.html
- Stored Routines and MySQL Privileges
  https://dev.mysql.com/doc/refman/8.0/en/stored-routines-privileges.html
- DECLARE ... HANDLER Statement
  https://dev.mysql.com/doc/refman/8.0/en/declare-handler.html
- Server Error Message Reference
  https://dev.mysql.com/doc/mysql-errors/8.0/en/server-error-reference.html

5 Exercise

Write stored routines for following task

1. List name of student from student table who takes course in given semester.
2. List all customer who done a payment with a given staff id.
3. List instructor's name and ID whose salary is less or more than a given amount to the average salary of all instructors in the university.
4. Find the name of all ingredients supplied by a specific vendor.

**********END**********